Fall 2015

Load Balancing for Entity Matching over Big Data using Sorted Neighborhood

Yogesh Wattamwar
San Jose State University

Load Balancing for Entity Matching over Big Data using Sorted Neighborhood

A Project
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfilment
Of the Requirements for the Degree
Master of Science

By
Yogesh Wattamwar
Dec 2015

The Designated Project Committee Approves Project Titled

Load Balancing for Entity Matching over Big Data using Sorted Neighborhood

By
Yogesh Wattamwar

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

Dec 2015

Dr. Duc Thanh Tran	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science
Dr. Suneuy Kim	Department of Computer Science

ACKNOWLEDGEMENTS

I would like to take this opportunity to convey my thanks to Dr. Tran, my project advisor, for introducing me to this interesting topic for the project that helps me to get to know more about the field of information retrieval. I got the chance to work on open-source software such as Apache Spark [3] and Hadoop [2]. Dr. Tran constantly provided me insightful feedback and his knowledge and experience in Big Data was very helpful to encourage me to go beyond what I could imagine. I am very grateful to him for all of that.

I also want to thank Dr. Thomas Austin and Dr. Suneuy Kim for investing your valuable time to review my report and being my committee members.

At the end, I would like to convey my thanks to my parents for their continuous support and encouragement during my studies and for giving me the chance to study in the U.S. They have been my source of inspiration throughout my journey.

ABSTRACT

Load Balancing for Entity Matching over Big Data using Sorted Neighbourhood

By Yogesh Wattamwar

Entity matching also known as entity resolution, duplicate identification, reference reconciliation or record linkage and is a critically important task for data cleaning and data integration. One can think of it, as the task of finding entities matching to the same entity in the real world. These entities can belong to a single source of data, or distributed data-sources. It takes structured data as an input and process includes comparison of that structured data (entity or database record) with entities present in the knowledge base. For large-scale entity, matching data has to go through some sequence of steps, which includes Evaluation, Preprocessing, Candidate calculation and Classification.

The entity matching workflow consists of two strategies: blocking (map) and matching (reduce). Blocking strategy termed as the division of a data source into partitions or blocks. Blocking is helpful to improve performance. Blocking achieves this goal restricting the set of similar entities in the same partition or block and then, comparing the same within blocks. The partitioning makes use of blocking keys and blocking keys are determined from entity's attributes. Partitioning helps to partition data into blocks. Values of one or several attributes form the blocking key. Mostly, the blocking key is concatenation of prefixes of these attributes.

The second part of the workflow consists of the strategy for matching. This aims to identify all matching entity pairs within the same partition. To find out matching result, one need to realize comparison result of the pair of entities. A

matching strategy can use several approaches for matching and can combine similarity scores to find if the entity pair is a match or not. The entity-matching model expects the matching strategy to return the list of matching pairs of entities.

Thus, by relating the structured data with their most apposite entity, entity matching tries to gain the maximum out of the existing knowledge base. One of the best solutions for Entity Matching would be Dedoop [4], which is Deduplication of Hadoop.

Cartesian product causes the workload due to execution with the time complexity of O (n^2) and to provide more time for matching techniques to maintain the quality, some load balancing techniques are necessary.

Even after the application of blocking, the task of matching i.e. Entity Matching can still be a costly task and can take up to several days for completion if running against large datasets. The MapReduce [2] programming model is perfect to execute EM in parallel. During execution, input file split into multiple parts or chunks. Then, map phase, multiple map tasks can read those parts in parallel, which are nothing but entities. During reduce phase, based on blocking keys, these entities are redistributed among several reduce tasks. This is helpful for grouping together entities with the same blocking key and can be helpful for the application of matching in parallel.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

Table 1: Acronyms [1]

Symbol	Meaning
SN	Sorted Neighborhood Approach
JobSN	Sorted Neighborhood with Extra MapReduce Job
RepSN	Sorted Neighborhood with Entity Replication
w	Window size
r	Number of reducers
m	Number of mappers
k	Blocking key value
p(k)	Partition prefix function for blocking key
HDFS	Hadoop Distributed File System
VM	Virtual Machine

CHAPTER 1

Introduction

Inspired by the research paper of Dr. Lars Kolb [1], Dr. Andreas Thor [1] and Dr. Erhard Rahm [1] about blocking using sorted neighborhood [1], this project aims to find out possible solutions for a parallel entity matching or resolution using the MapReduce programming model with Apache Spark [3] and Hadoop [2].

Entity matching is a data-intensive task and benefited from cloud computing due to its scalability in terms of performance. The entity resolution used for determining entities from a given source, which are a close match to the given object. This provided object is mostly a real world object. Therefore, it is very important for

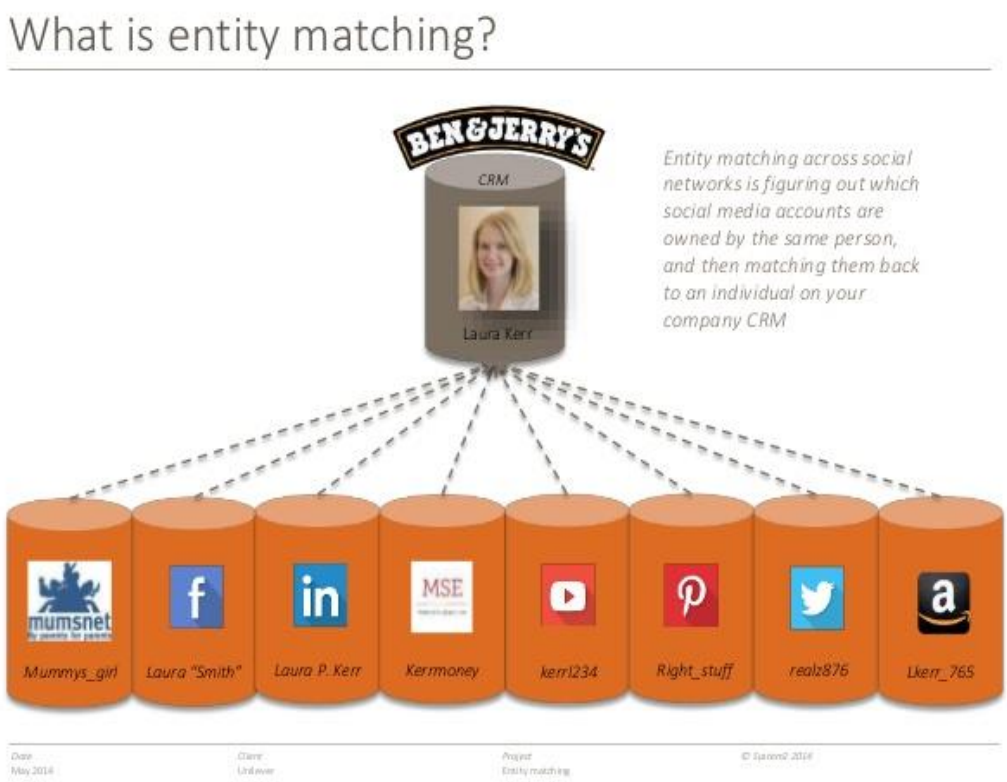


Figure 1 : Entity Matching [9]

data quality and data integration

Currently, as Figure 2 demonstrates a general entity matching workflow. Entity matching aims to check the given real world entity against the available list of entities and returns all those with the most approximate match.

Among all proposed approaches and frameworks for Entity Matching, the standard approach for matching n input entities is comparison of all entities with each other, which is the Cartesian product (n * n) of all input entities. Time complexity for such approach is O (n^2). For very large datasets, this causes intolerable execution times. In this case, so-called blocking techniques are required to improve the performance. Blocking help by reducing the number of entity comparisons and makes enough room to maintain match quality at the same time. Sorted neighborhood (SN), as the name suggests, all entities sorted using the blocking key and then, compares entities within a predefined specific range, which referred as a distance window w. In the Sorted Neighborhood approach, the complexity of matching is O (n * w) which is very less compared to the quadratic complexity of O (n^2).

In this thesis, I find the approach for load balancing with the use of MapReduce [2] and Spark [3] for the parallel execution of sorted neighborhood [1] blocking and entity resolution. I am aiming for an efficient entity matching implementation using a combination of blocking and parallel processing. This can be helpful for processing very large datasets. The proposed approaches makes use of partitioning techniques that are specific to the MapReduce [2] programming model with a correct implementation of the sliding window approach for comparison of entities. My contributions towards the thesis are:

• Demonstration of how the spark MapReduce [2] model applied for the entity matching. This makes the process run in parallel and divides in two phases: blocking and matching.

• Demonstration of how the spark MapReduce [2] model applied for the execution of two suggested approaches Sorted Neighborhood with an extra MapReduce job and Sorted Neighborhood with Entity Replication for entity resolution workflow in parallel. This again divided in two phases: blocking and matching.

• Identification of major challenges and propose an approach to improve load balancing during Neighborhood Blocking on spark MapReduce. Besides multiple spark-MapReduce [2] jobs, the approach use efficient Partitioner [2] to repartition the data in case unequal distribution of data to balance the load across nodes.

I evaluate my approach against given approaches in paper [1] with higher and lower data skew and demonstrate its efficiency in comparison to the sequential approach with effect of the factors like, data skew and the sliding window size with different configurations for the number of nodes.

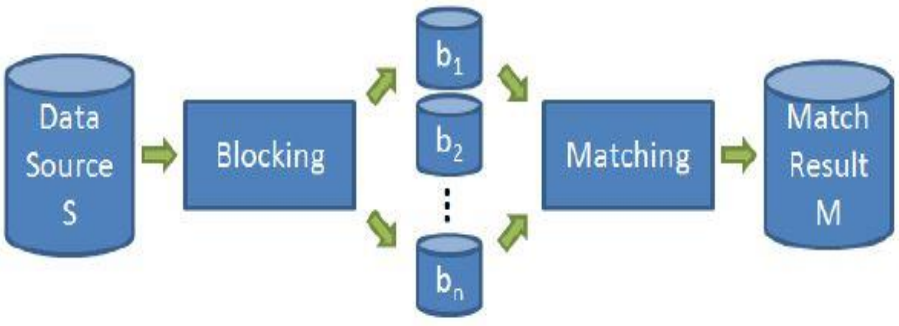


Figure 2 General Entity Matching Workflow [4]

CHAPTER 2

Related Works

This chapter consists of related work towards entity matching or entity resolution. In his paper on entity matching [1], Dr. Kolb and his colleagues described about the Sorted Neighborhood approach for entity matching which can be implemented using MapReduce programming model. Entity matching is comprised of two phases if one wants to mold it in the MapReduce [2] model where the first phase is blocking which is a map task and the second phase is matching which is a reduce task. MapReduce does not provide direct provision to compare entities in different blocks during reduce function. They investigated that simple MapReduce algorithm is not enough to execute Sorted Neighborhood, as it does not compare boundary entities with each other as per window size. Here, boundary entities are those, which are in the range of first and last w – 1 where w, is window size entities for all partitions except for first and last partition. For the first partition, last (w – 1) entities are boundary entities and for last partition, first w–1 entities are boundary entities. Therefore, Dr. Kolb and his colleagues investigated two approaches to implement Sorted Neighborhood, which Sorted Neighborhood with an extra MapReduce job and Sorted Neighborhood with Entity Replication to achieve entity resolution with parallel blocking with Sorted Neighborhood using MapReduce architecture. In this paper [1], there is a scope for improvement of load balancing when data skew is very high. In both approaches, when skew is very high execution time is very long which hampers the performance.

CHAPTER 3

Problem Definition, Existing Solution and Proposed Solution

If the user wants to search for an entity with the name “Ryan McCarthy” from the given list of entities then, entity is determined from the available list of entities. System then, search for all entities with the name “Ryan McCarthy” and matches it against few more measures to get the most exact result. Entity Matching techniques usually make a match decision based on comparison of pairs of entities and evaluation of multiple similarity measures. If we decide to follow the naïve approach, then, we can observe the Cartesian product of all input entities. Therefore, it is very inefficient for large datasets due to this resulting quadratic complexity of O (n^2). Performance is worse even if we execute it on the cloud environment. Adopting blocking techniques is the common approach to improve efficiency. In blocking, we semantically group similar entities based on blocking keys derived from entities’ attributes and restrict entity comparisons to entities from the same block and reducing the search space.

In his paper [1], Dr. Kolb and his colleagues followed the Sorted Neighborhood algorithm which reduces the quadratic complexity of matching to O (n * w) but again it also need the sorting to be done, which adds a complexity of O (n * log n). This complexity is more in sequential terms and execution time is linear. By using MapReduce programming model, this algorithm comprised of two phases. One is blocking during the map phase as described earlier and matching during reduce phase. Two approaches suggested are workable and provide better performance over huge datasets. There is no provision for load balancing when data skew is high. In case of higher data skew, execution time is more and window size w has to be increased.

The proposed solution check if there exists a partition with at least 3 times more than average partition size before running reduce operations over map output. If such partition exists then, we repartition that block in equal sized block equal to average partition size to balance the load across all nodes. Then, to complete Sorted Neighborhood implementation, we cover new boundary entities using Sorted Neighborhood with an extra MapReduce job approach and compare newly identified boundary entities.

If we take an example of 50000 entities and after map phase, there are 4 partitions with sizes 10000, 35000, 5000 and 10000 entities. In this case, we will figure out partition 2 has most of the data and that need to be redistributed in 3 more blocks of size 12500, 12500 and 10000 and adding the boundary entities to the newly added partitions. Once done, reducers run in parallel, which are designated matching jobs.

CHAPTER 4

Overview

4.1 MapReduce

MapReduce [2] is a programming paradigm, which allows scalability on a very large scale. It allows programmers to process and produce huge data sets. This programming model invented in Google in 2004. User specifies two operations in MapReduce [2], a map function and a reduce function [8].

MAP
• Processing a key/value pair to produce intermediate key/value.
Reduce
• Concatinate all intermediate values with proper association of key.

Table 2: MapReduce Key/Value pairs [8]

The code, which we write using MapReduce [2] parallelized automatically. In order to understand how the model works we can go over the steps on which it executes.

1. Given input splits into multiple partitions, output key/value pairs generated.
2. Then the user needs to write the code for map and reduce functions.
3. Map uses the input as the key/value pair [8] and generates an intermediate key/value pair.

4. The MapReduce library then, groups them and generate an intermediate key say **"Intermediate key"**
5. This key is then, passed to the second function.
6. The input to this reduce function is **"Intermediate key".**
7. As the name suggests it reduces the value to smaller set of values.
8. It is important to point out; there exists an iterator, which give **"Intermediate key"** to reduce function.

In order for a programmer to write code for a map or reduce operation, the user needs to know some functions, which facilitates with a job. The functions have particular functionality and can take specific a set of inputs. Every function we write is to yield a specific output. Four very important functions of map reduce can be viewed below. The table below mentions the function name, its functionality, acceptable input and output.

The first step, in execution of a MapReduce [2] program is the user's code. The user writes a code. This code consists of two functions, Map and Reduce. This code passed to master node and then this master node assigns map function to one worker and reduces to another worker node. The input over which the MapReduce will be working split into smaller chunks. All these splits worked on in parallel. This happens after map function invocated. Once reduce function invocation is done, then, grouping of similar keys performed and sorting based on key values. Here, partitioning can play a vital role where partitions assigned using partitioning functions. This is one of the functions mentioned above. The user [8] specifies this number.

Mapper	Reducer	Partitioner	Combiner
data--> key/value pair	key/vlaue --> output	key/value distribution in partitions.	combines partitions
:: *Input* :: Input data	:: *Input* :: key, list of values	:: *Input* :: key, value	:: *Input* :: key, list of values
:: *Output* :: key, value	:: *Output* :: output object	:: *Output* :: no of partitions.	:: *Output* :: key, reduced value list

Table 3: Four Functions of Hadoop [8]

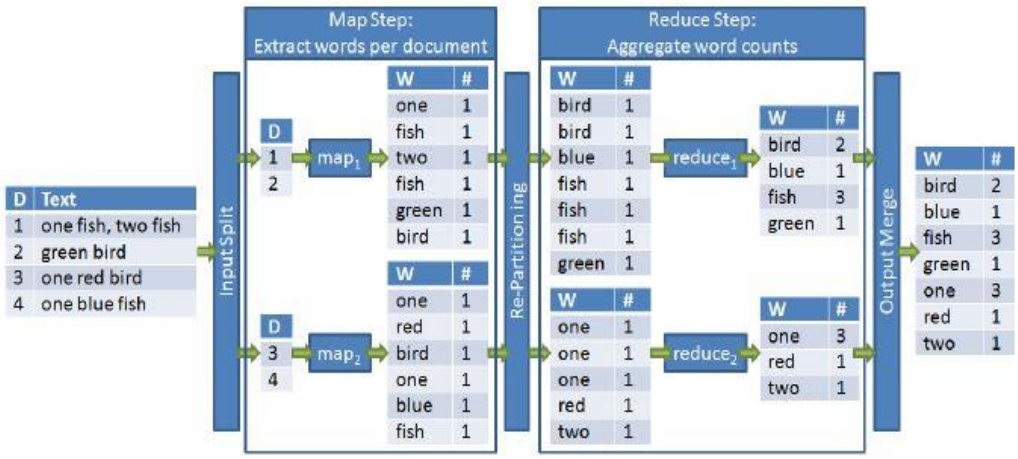


Figure 3: Word count program workflow with MapReduce [1]

Figure 3, depicts a simple example of how data flow takes place in MapReduce operation. It shows how the MapReduce program for the word count works. This program counts the number of word occurrences across multiple documents. The number of mapper here is 2 as we can clearly see in the figure. The input data has 4 input values which are distributed between the 2 mappers present in this system. Thus, we see each mapper gets 2 documents. This is a simple example, but in reality, we do this over a large data set and in parallel. Mapper id coded to generate the list of words and pair it with a key. Now, it hands over to reducer. Let us say all words from A → M sent to reduce 1 and rest to 2. This reduce function combines the word occurrences and produces final output, which says how many times each word, occurred. Figure 4, below explains how reduce

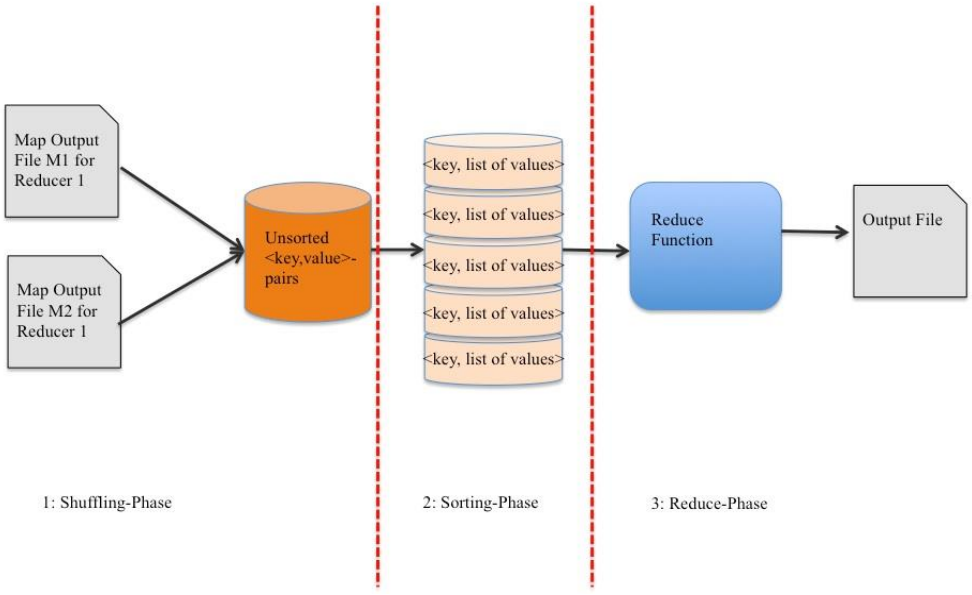


Figure 4: Overview of Reduce phase [2]

operation is performed in the MapReduce model.

The MapReduce-algorithm, Spark, and Apache Hadoop are powerful and cost-effective approaches to process large amounts of data in a distributed way. Hadoop [2] helps the user to handle unstructured data and helps to provide stability throughout different platforms and different instances. In both Spark and Hadoop, the load of work distributed throughout a cluster of instances.

4.2 Sorted Neighborhood Approach with the MapReduce [1]

This approach is a very popular among all blocking approaches. A blocking key K, which can be the concatenation of prefixes of a few entity attributes, is determined for each of n entities. Afterwards this blocking key sorts the entities.

During the next phase, a window of fixed size w then, used by sliding for comparison over all the sorted records and all entities within the range of that window compared during each slide of the window.

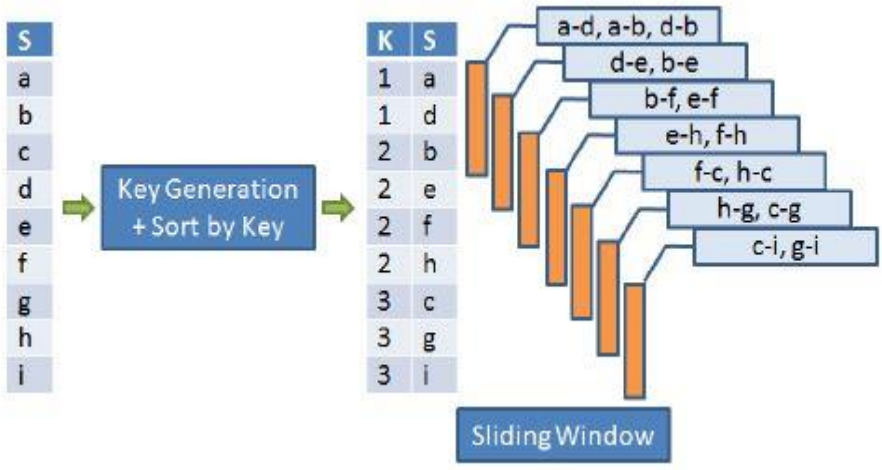


Figure 5: SN Execution with Window size 3 [1]

Figure 5 shows an example of sorted neighborhood approach execution. Window size is defined as (w = 3). The input consists of 9 entities. Blocking keys are determined as (1, 2, or 3). In sorted neighborhood approach, first all entities are sorted using their blocking keys which are 1, 2 and 3 in this example and entities are from a to i. The window for comparison starts with the first block consisting of the first 3 entities resulting in three pairs for later comparisons. The window is then, moved forward to cover the next block, which includes entities d, and b from the previous window and entity e from the current window. This adds two more pairs for comparison. This window moved forward until it has reached the end of the input set. All generated pairs by the sliding window process listed in Figure 5. On an average, the total comparisons performed by sorted neighborhood is (w - 1) * (n - w/2).

This approach helps to reduce the complexity from O (n^2) to O (n * log n) + O (n) to determine the blocking key and sort and for matching it is O (n * w). Hence, it is possible to match large datasets and the runtime controlled with help of window size w. In addition, it can compare different entities with various attributes forming blocking keys. Hence, choice of the blocking key does not affect this approach.

To check the accuracy, we can execute the sorted neighborhood approach repeatedly using different entity attributes to form blocking keys. As a result, this reduces the effect of selecting poor blocking keys, which may be due to data and at the same time, maintains the linear complexity against the matching approach.

In this way, this approach proves to be more robust against load balancing due to the linear complexity. Unlike other blocking techniques, in the sorted neighborhood, it is not mandatory for a matcher to compare those entities that share the same blocking key. For example, in Figure 5, entities b and d possess different blocking keys, and according to the sorted neighborhood approach, there should be comparison between these two entities. On the other hand, MapReduce [2] has a concept that input partitions for map processed independently, which, contradicts to the sorted neighborhood paradigm of comparing entities with different blocking keys. However, in MapReduce [2], this helps for an efficient parallelization model, but as a mapper does not have access to the partition of other mappers, it is a challenge to group together boundary entities within a distance of w. For comparisons, the sliding windows approach can lead to heavy overlapping of entity set. The MapReduce-based approach is applicable here, but requires expending unnecessary resources. If we go by numbers, then, all entities that appear in w blocks, appears the same number of times in the output of map operation. During reduce phase, there can be a problem of duplicate pair generation due to the overlapping blocks.

Therefore, Dr. Kolb and his colleagues target a more efficient solution for implementation of sorted neighborhood using MapReduce and, thus, adapt the approach that for each input entity, the map function used for determination of the blocking key independently. The output of map operation then, provided to available reducers. Reduce function implements the sliding window approach, and then, reducers are assigned to run the same for all available reduce partitions.

For example, when we have two reducers, the first reducer gets all entities with the blocking key, (k <= 2) and entities with the blocking key, (k >= 2) given to the second reducer. This scenario brought two challenges to implement sorted neighborhood approach using MapReduce.

Sorting across Partitions: As, the sorted neighborhood approach assumes a sorted list of entities and sorting done based on their blocking keys. Due to this, it becomes necessary that repartitioning must preserve this order. The sorting should span over all available partitions where the output of map function has to assure that all entities assigned to reducer R_y have equal or a smaller blocking key compared to all entities of reducer R_y+1. Due to this, each reducer can apply the sliding window approach to all partitions.

It utilizes an appropriate user-defined function f to calculate partition prefix, which used for redistributing data among available reducers in the map phase. This generated blocking key k helps to redistribute data later. Here, f is a function (f: k → i with (1 <= i <= r)) [1] where r refers to the number of reducers. This function f where f(k1) >= f(k2) if k1 >= k2 [1], assigns entities with smaller or equal, the blocking key than any entity processed by reducer (i + 1), to current reducer i.

As blocking keys usually derived from entity attributes, possible blocking keys guessed or known beforehand for a given dataset. Hence, in real time

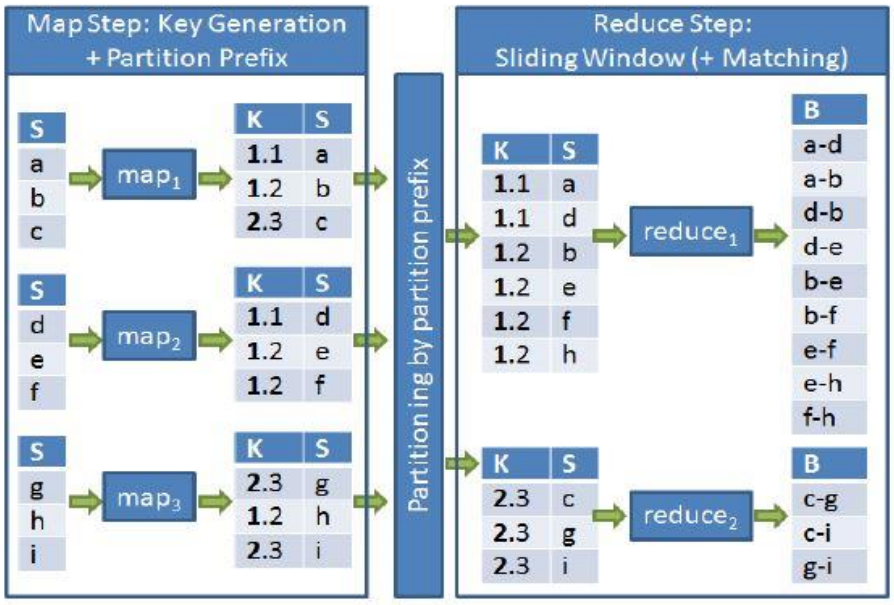


Figure 6: Execution of SRP [1]

implementations, range partitioning functions f would be helpful.

Figure 6 illustrates the execution of SRP with (m = 3) mappers and (r = 2) reducers. Here, entities used are same as the previous example. In this given example, the function f returns 1, if (k <= 2), else returns 2. This value returned by the function f added as a prefix to the blocking key k generated during the map function for each input entity. As a result, combined key value is 1.2 where 1 is partition prefix and 2 is the blocking key. Then, depending on the partition prefix of the key, we distribute (key, value) pairs by using partitioning.

For example, reducer 2 will receive all the keys with prefix 2. Here, input partitions actually sorted using the blocking key. All keys for a reducer should

start with the same partition prefix. Therefore, the sorting of the keys actually done using the blocking key, which comes after prefix.

Once the sorting done, the reducer is free to run the window with the predefined size for comparison, and, hence, as a result, it generates the list of all matching pairs of entities. One issue with this approach is it does not allow comparison of boundary entities of partitions. It does not compare the last entity in a partition with next (w - 1) entities. In the same manner, it does not allow the first entity to compare with last (w – 1) entities of the previous partition. For example, SN approach must identify the pair of entities h and c, and framework cannot generate as c and h assigned to different reducers. Therefore, considering the configuration with r reducers and a window of size w, SRP has missed ((r - 1) * w * (w - 1)/2) boundary entity pairs.

Boundary Entities: Boundary entities defined as entities, which fall in the range of first and last (w – 1) entities in all partitions except first and last partition. In the first partition, last (w – 1) entities are boundary entities and in the last partition, first (w – 1) entities are boundary entities. As per standard SN, these entities compared with entities corresponding to them in sequential manner. When we use it in MapReduce paradigm, these entities are ones with first (w – 1) entities of next partition.

Additional or Extra MapReduce job for Boundary Entities:

To resolve the problem of boundary entities, Dr. Kolb and his colleagues introduced this approach. This approach utilizes the previous approach SRP and assigns a second MapReduce job to deal with the boundary entities. After which the sorted neighborhood result is completed. MapReduce provides sorted partitions to the reducer and JobSN uses this fact. Due to this fact, during the local execution, it is easy for a reducer to find out the first and the last (w – 1) entities. Those entities have their related entities in previous and next partitions, in simple terms, the last w - 1 entities of a reducer related to the first w - 1 entities of the next reducer. Generally, all reducers except first and last reducer provide the first (w – 1) entities and last (w – 1) entities. The first reducer provides only the

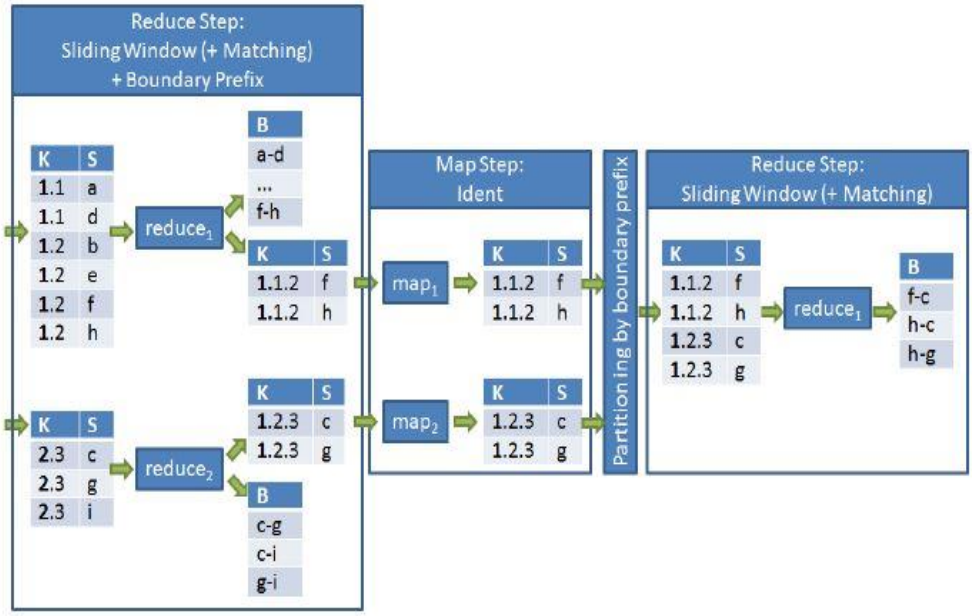


Figure 7: SN Execution with an extra MapReduce job [1]

last (w – 1) entities and the last reducer returns only the first (w – 1) entities.

Figure 7, illustrates a sample execution of JobSN approach. It uses the same data of Figure 6. The map operation of the first job is the same as the previous

approach of Figure 6 and skipped in this figure. The figure shows an extension of reduce step with an additional output. As described above, the reducer provides the first (w – 1) entities and last (w – 1) entities with the output list of matching pairs.

Here, related boundary elements identified using a boundary prefix that defines the boundary number. Since the last (w – 1) entities of reducer 1 < r refer to the first boundary, for the last (w – 1) entities, keys are prefixed with 1. In the same manner, the first (w – 1) entities of the reducer 2 also relate to the first boundary. Hence, the keys of the first (w – 1) entities of reducer 2 > 1 prefixed with (2 – 1 = 1).

The first reducer in the example of Figure 7 adds a prefix of 1 to last entities (h and f) and the second reducer adds a prefix of 1 to first entities (g and c), too. Hence, data lineage reflected in the key: The raw version of the blocking key for entity c has value 3, and it assigned to reducer number 2, and it related to boundary number 1. During the last MapReduce job, the map functions does not make any changes to the input data and reduce function is straightforward.

Available reducers receive distributed output of the map function based on the boundary prefix. Here, reduce job filters pairs of entities identified in the first MapReduce job and applies the sliding window on the rest. For example, this pair (f, h) already determined during the first MapReduce job and hence, skipped in the second job. At the cost of an extra MapReduce [2] job, this approach generates the complete result of the sorted neighborhood approach. There is an

expected overhead for an extra MapReduce job, which should be acceptable in this approach.

Replication of Boundary Entities:

The replication of boundary entities approach introduced to implement the sorted neighborhood without the extra MapReduce job overhead. Alike the previous approach, this approach utilizes SRP technique and extends the same by the idea that the relative boundary entities appear in the output of reducer. To achieve this, each reducer will have the first (w – 1) entities of the next reducer at the end of its input.

However, the MapReduce paradigm does not support the fact that reducers can access mutual data. MapReduce only provide options to control replication of data within the map function. Therefore, this approach make changes to the existing map function of SRP to replicate an entity that sent to both the current reducer and its successor. Thus, map function identifies the (window size – 1) entities possessing the highest blocking key for all partitions except for the last reduce partition. Therefore, map function first processes all entities and output them. Once done, it appends all the entities, which identified as boundary entities at the end. The blocking key (k) and a partition prefix f (k) determines an entity key. Here, an additional boundary prefix added to differentiate between actual entities and replicated boundary entities.

For original entities, a partition number is boundary prefix, which means the combined key is p (k).p (k).k. For replicated entities, a partition number of next

reducer is the boundary prefix, which means the combined key is (p (k) + 1).p (k).k.

Figure 8 shows execution of this approach. The example uses 2 reducers and window size 3. Therefore, all the mappers identify 2 entities (w – 1 = 2) with the highest key of the first partition. The output of each map function divided virtually into two parts. The first part, which consists of original entities with the partition, prefix same as the number of current reducer partition and the second part which, consists of replicated entities with boundary prefix as its own prefix.

Alike previous approaches, available reducers receive distributed map output depending on the boundary prefix of entities. The replicated boundary entities appear at the beginning of each reducer input. This is because MapReduce provides a key based sorted list as input to reduce functions and as the structure of the combined key has all prefixes. The replicated boundary entities have a smaller partition prefix but boundary prefix is same. Here, reduce function implements a little different sliding window with the criteria that pair of entities generated as an output contains at least one actual entity from the partition.

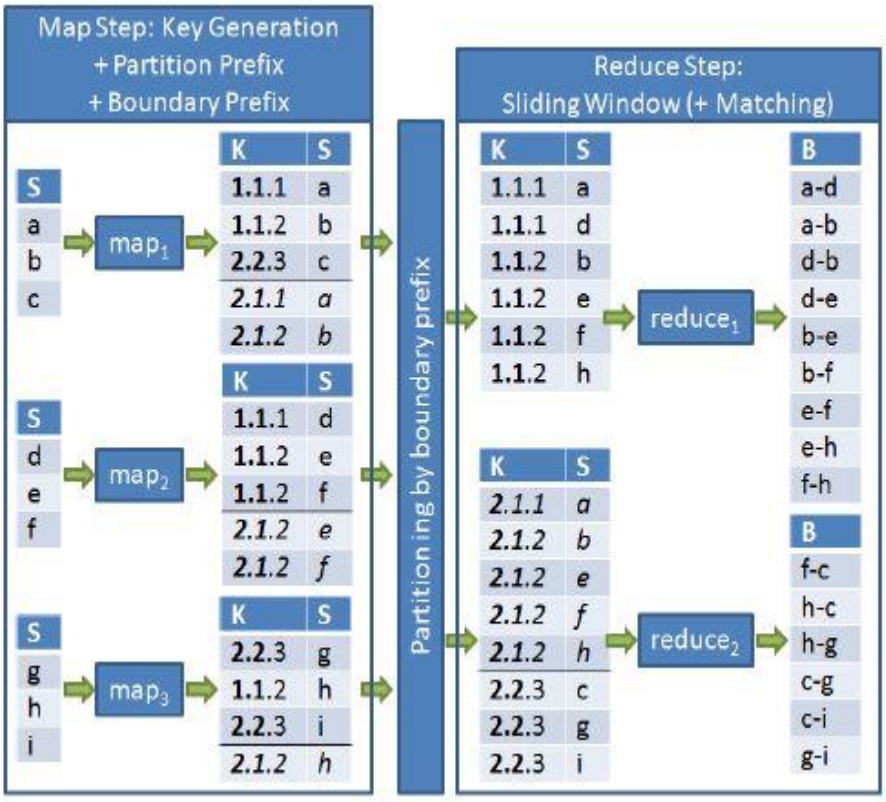


Figure 8: SN Execution with Replication of Boundary Entities [1]

In the above example, due to the addition of boundary entities, the second reducer gets a larger partition of input. During the process, it finds (w – 1 = 2) the highest entities (f and h) and ignores the rest of the replicated entities. The result is the complete output of sorted neighborhood approach for entity matching. This approach aims for sorted neighborhood execution within one cycle of MapReduce job with the expense of data replication. Each map function identifies based on its local data and replicate possibly relevant entities. Except the last partition, for all partitions each mapper replicates w - 1 entities.

Therefore, the total number of such replicated boundary entities can be (r - 1) * (w - 1) [1]. However, this number is independent of the size of input entities but might be small for larger datasets.

4.3 New Approach: Extension of SN with extra MapReduce Job

JobSN and RepSN does not have any load balancing strategies and can perform very bad in case of large data skews. There is no check provided on partition size before proceeding with reduce operation. If the data skew is high then, the execution time may increase due to one reducer as there might be a reducer running for a longer duration for one particular partition, which has occupied almost all of the keys. In such scenario, data skew is very high and load balancing is required. First, let me help you understand data skew in detail.

Data skew:

Data skew means a non-uniform distribution in a dataset. Data skew impacts directly on parallel execution of complex database queries when there is a poor load balancing which leads to high response time for query.

There are primarily two types of skew, **intrinsic skew** or **attribute value skew** which is referred as a skew intrinsic in the dataset. Second type of skew is **partition skew,** which occurs on parallel implementations. Even when input data is uniformly distributed, the workload not evenly distributed between all nodes. Partition skew classified in four more types.

Tuple placement skew which is introduced during initial partitioning of data, selection skew which is introduced during selection of different select predicates on each node, redistribution step skew which is introduced during the

redistribution phase between two operators and join skew is introduced during the join selectivity phase which may vary between nodes.

Algorithm:

This new approach extends JobSN by adding a check if there is any partition of size more than 3 times of average partition size of current process after repartitioning. If there is such a partition present then, that partition is marked and repartitioned one more time with new Partitioner [2], which divides the partition in new small partitions of the size equal to average partition size. In this way, during the reduce phase these new partitions will balance the load which is being transferred to all nodes. Here, one thing I should take care of is boundary entities comparison but this will be taken care by extra MapReduce job as mentioned earlier in JobSN approach. By implementing this, there is no harm to sorted neighborhood approach as the boundary entities created recently by second repartitioning, covered by extra MapReduce job and hence, the implementation is complete.

Algorithm implemented to handle Data skew problem (Extension of JobSN) :

Map Phase I

1. Key assignment based on composite key of two columns.
2. Partitioning is done based on partition prefix of composite key using a partitioner.
3. **If partition size > 3 * (average partition size) then, repartition using a new partitioner.**
   if(partition_size > 3 * (average partition size))
   {
   --repartition using another partitioner function for larger partition
   }

Reduce Phase I

4. In Reduce task run a window of size w in each partitions and identify boundary entities.

Map Phase II

5. JobSN gathers all the boundary entities for comparison.

Reduce Phase I

6. In Reduce task run a window of size w in each partitions

Figure 9: Algorithm for New Approach (Extension of JobSN)

This approach works better when there is very large data skew and also, helps avoiding the problem of memory bottleneck in case of large partitions of data up to some extent. It does not affect the performance in case of normal execution.

CHAPTER 5

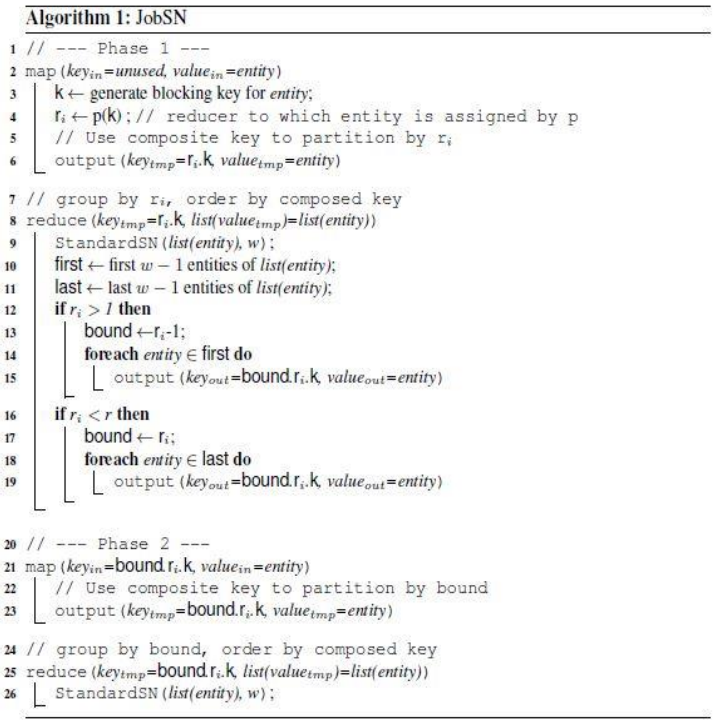
Implementation

This chapter provides information about algorithm followed for implementation of the baseline, dataset used for testing and tools used for the implementation. Here, I will jump directly on the implementation of JobSN and RepSN, as both are the extension of Sorted Reduced Partitioning approach. Followed by these two approaches, I will discuss the implementation of new suggested approach, which is again an extension of JobSN approach.

5.1 JobSN

As discussed earlier, JobSN utilizes SRP and completes the sorted neighborhood implementation by detecting boundary entities during reduce phase and then, running an extra MapReduce job over boundary entities to run a sliding window for matching again.

As described in algorithm, there are two MapReduce phases in the algorithm for JobSN. The first map operation calculates the key with the help of blocking key and partition prefix. Partition prefix derived from the function f (k) and then, the blocking key added to it. After partitioning, for each partition except first and last partition, during reduce phase a sliding window of size w executed and at the same time, partitions assigns first and last (w – 1) entities a boundary prefix which is a partition number itself. For the first partition, last (w – 1) entities assigned a boundary prefix and for last partition, first (w – 1) entities assigned a boundary prefix. During next MapReduce phase, these entities are collected in map phase and a sliding window of size w is executed during reduce phase in every partition after



Algorithm 1: JobSN

```
1  // --- Phase 1 ---
2  map (key_in=unused, value_in=entity)
3      k ← generate blocking key for entity;
4      r_i ← p(k) ; // reducer to which entity is assigned by p
5      // Use composite key to partition by r_i
6      output (key_tmp=r_i.k, value_tmp=entity)
7  // group by r_i, order by composed key
8  reduce (key_tmp=r_i.k, list(value_tmp)=list(entity))
9      StandardSN (list(entity), w) ;
10     first ← first w − 1 entities of list(entity);
11     last ← last w − 1 entities of list(entity);
12     if r_i > 1 then
13         bound ←r_i-1;
14         foreach entity ∈ first do
15             output (key_out=bound.r_i.k, value_out=entity)
16     if r_i < r then
17         bound ← r_i;
18         foreach entity ∈ last do
19             output (key_out=bound.r_i.k, value_out=entity)
20 // --- Phase 2 ---
21 map (key_in=bound.r_i.k, value_in=entity)
22     // Use composite key to partition by bound
23     output (key_tmp=bound.r_i.k, value_tmp=entity)
24 // group by bound, order by composed key
25 reduce (key_tmp=bound.r_i.k, list(value_tmp)=list(entity))
26     StandardSN (list(entity), w) ;
```

Figure 10: Algorithm for JobSN [1]

repartitioning.

5.2 RepSN

RepSN utilizes SRP, completes sorted neighborhood implementation by detecting the boundary entities during map phase, and then, adds boundary entities to the end of partitions so that boundary entities covered when we run a sliding window for matching again.

As described in algorithm, RepSN finds entities with the maximum blocking key during map phase in each partition at the local level. Once done, it replicates and adds a boundary prefix to the blocking key of replicated entities with the value (partition number + 1). For actual entities, the value of boundary prefix will be same

as the partition number. During reduce phase, a sliding window of size w executed for actual entities in a partition. Hence, this completes the implementation of sorted

Algorithm 2: RepSN

```
1  map_configure
2      // list of the entities with the w-1 highest
3      // blocking keys for each partition i<r
4      foreach i ∈ {1, ..., r − 1} do
5          rep_i ← [];
6  map (key_in=unused, value_in=entity)
7      k ← generate blocking key for entity;
8      r_i ← p(k); // reducer to which entity is assigned by p
9      bound ← r_i;
10     if r_i < r then
11         if sizeOf(rep_ri)<w-1 then
12             append(rep_ri, entity);
13         else
14             min ← determine entity from rep_ri with smallest blocking key;
15             k_min ← blocking key of min;
16             if k >k_min then
17                 replace(rep_ri, min, entity);
18     // Use composite key to partition by bound
19     output (key_tmp=bound.r_i.k, value_tmp=entity)
20 map_close
21     foreach i ∈ {1, ..., r − 1} do
22         r_i ← i;
23         bound ← r_i +1;
24         foreach entity ∈ rep_i do
25             // prefix key with r_i+1 to assign replicated
26             // entities to succeeding reducer
27             output (key_tmp=bound.r_i.k, value_tmp=entity)
28 // group by bound, order by composed key
29 reduce (key_tmp=bound.r_i.k, list(value_tmp)=list(entity))
30     remove all entities with bound ≠ r_i from the head of list(entity) except the last w − 1;
31     StandardSN (list(entity), w);
```

Figure 11: Algorithm for the RepSN Approach [1]

neighborhood.

5.3 New Approach

New approach is an extension of JobSN algorithm where before reduce step and after grouping of keys performed we check for partition size. For any partition, if the partition size is 3 times more than average partition size, then mark that partition and repartition the data by using a new Partitioner [2] function, which breaks that partition in furthermore partitions of average partition size. Figure 12 shown below gives better idea about the algorithm.

After step 7 of grouping entities or repartitioning, I am providing a check for the size of partitions and consequently, data skew if higher enough to damage the performance. If I find out any partition has large data as compared to other partitions, then, I am repartitioning the data using a new Partitioner [2] logic, which has a check for the partition number and assigns the prefix as per count of records during partitioning. If we consider the case of boundary entities, which created after this partitioning, then, the second MapReduce job will compare those entities and completes the implementation of sorted neighborhood.

This algorithm helps handling the problem of data skew by repartitioning the data for larger partitions and if there is no large partition available then, execution completed in almost the same time as done in JobSN approach, which is very effective approach in a way. This may also help to avoid memory bottleneck issues due to the division of large data partitions to smaller partitions.

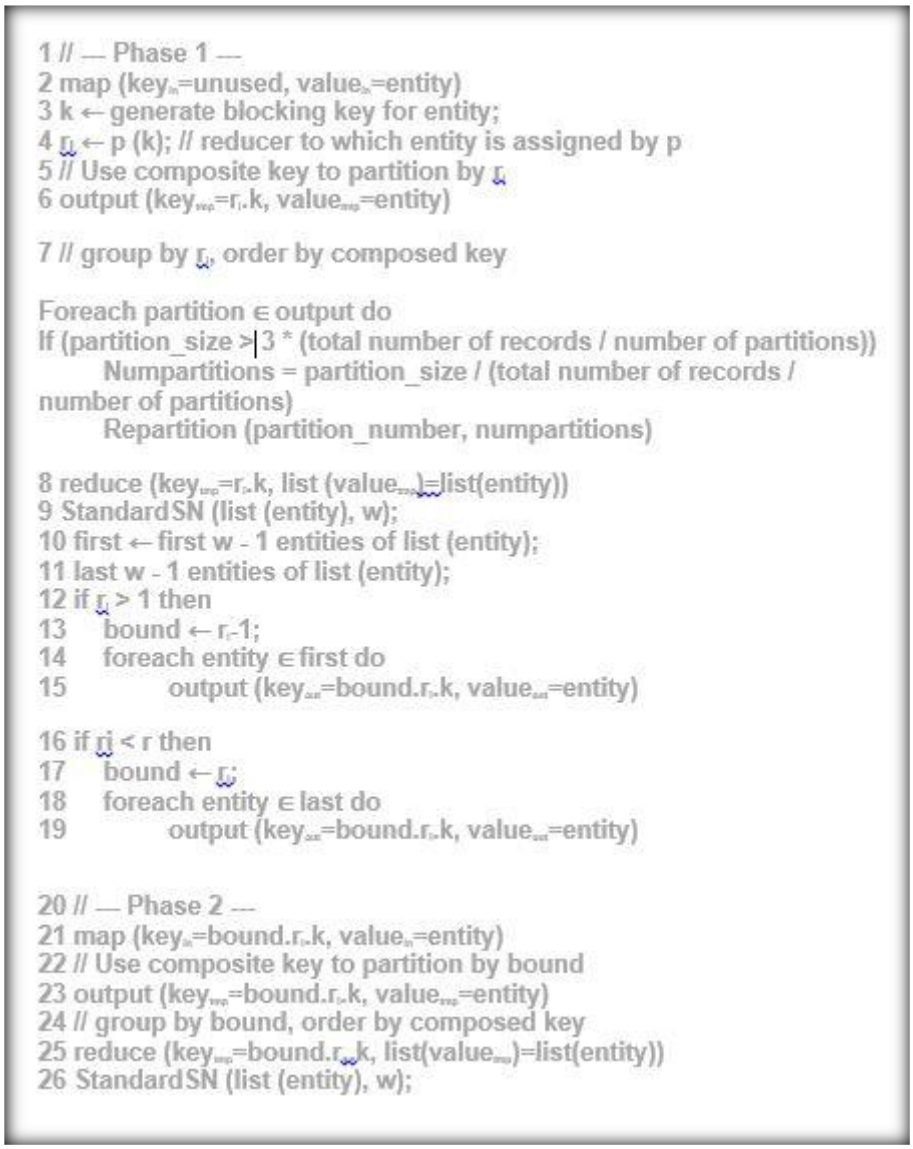


Figure 12: Algorithm for New Approach

5.4 Technologies Used

This project built upon Apache Spark using Java and Apache Hadoop with MapReduce programming model. Modern MapReduce implementations like Hadoop offer the user a large variety of possibilities to implement user-defined functions. The uneven distribution of the data may lead to a longer execution time in many situations. One challenge is to evenly distributing the workload among

the clusters. There are many examples of geographic data, skewed in nature and balancing the load may become necessary in such cases. For some applications, these unequal compositions of data may cause longer execution times. We can detect this uneven distribution due to the input data and prevent the same.

A strategy to balance the load of work can be crucial. Spark [3] provides faster execution than Hadoop [2]. Spark [3] can be 100x faster than Hadoop [2] for processing large-scale datasets.

Spark currently a record holder for on-disk sorting on large-scale data and is very fast on processing when data is stored on disk. It has APIs, which are easy-to-use and helpful to operate on very large datasets. Spark combined with other APIs to create complex workflows and has standard libraries that increase developer productivity.

Technology	Reason
HDFS	As a primary holder of the knowledge base upon which the big data processing layer will perform search and computation processes. I am planning to change some processing queries to match it to a cluster-based algorithm to improve speed. e.g. Dedoop
Apache Spark	As the core search and analytics engine. Due to its efficient in memory processing, Apache Spark appears as a fitting solution for the text oriented solution in consideration. Apache spark is itself good with speed and combined with rule-based algorithm.
Core Java	To test the program execution speed under all circumstances using basic data structures, and other algorithms to help giving an analysis for further proceedings.

Table 4: Technologies Used

CHAPTER 6

Experiments and Results

6.1 Experimental Setup

I have carried out my experiments on a spark cluster of 3 nodes with two cores (1 master and 2 slave nodes) using VM fusion and Ubuntu 14.04. Each node has 2 GB memory and operating system is a 64-bit Ubuntu 14.04 with Java 1.6. On all nodes, I am running Spark 1.2.1 with Hadoop distribution of version 2.4. Each node runs at most two worker instances, each with a memory of 1 GB.

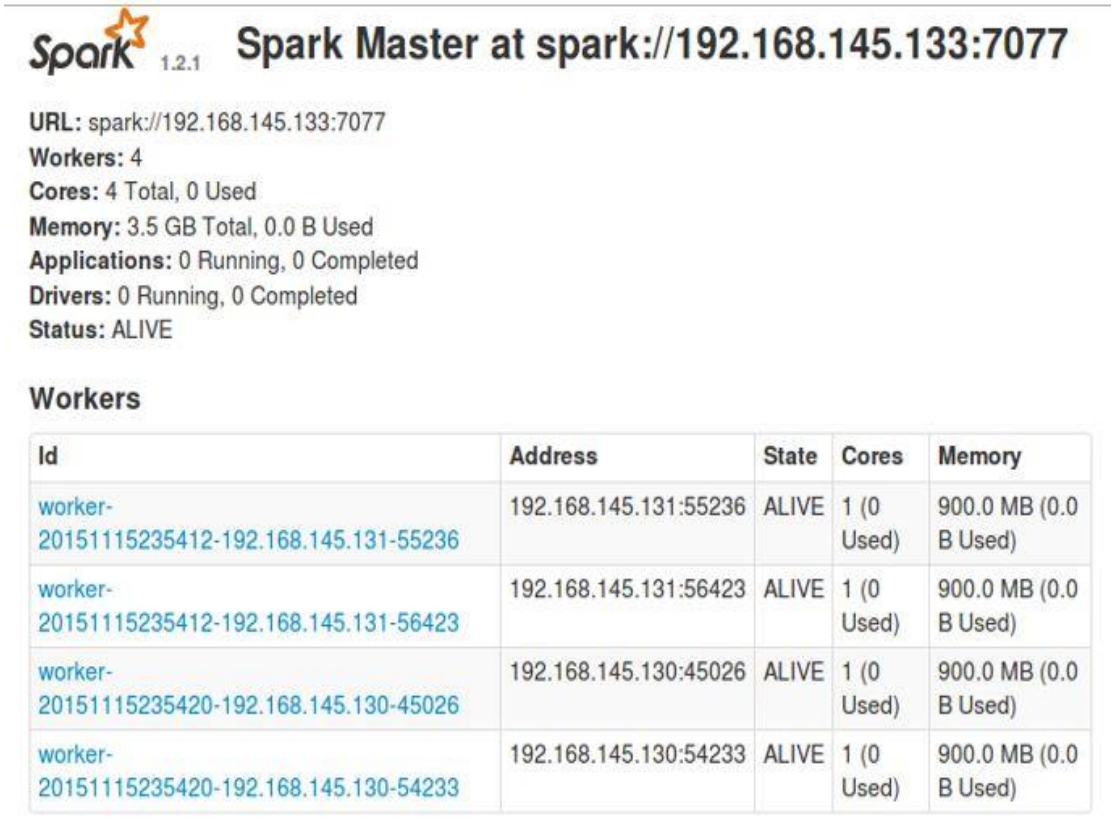


Figure 13: Spark Experimental Cluster Setups

I used Spark's text file with native block compression to serialize input file. Entity pairs with similarity score of 0.8 or more, referred as matches. Dataset used was

International Aiding and consists of approximately 1 Million records. Another dataset used was MIMIC 2, which I used to validate the results. To group similar entities into blocks I used the lower-case country values and currency as the blocking key, which we consider as a composite key.

6.2 Evaluation and Results

I evaluated Sorted Neighborhood in three critical factors

1. Window size
2. Configuring number of maps and reduce tasks
3. Number of available nodes in clusters
4. Degree of Data Skew

Window size and Configuring number of maps and reduce tasks:

I used the same defined function to determine partition prefix in each experiment to make sure that different numbers of mappers and reducers compared on the same basis. I evaluated the runtime and the relative speedup using 2 different window sizes of 10 and 100. The additional MapReduce job of JobSN and new approach was executed with one reducer (r = 1). Partitioner [2] divides or partitions the set of entities into 10 blocks and tries to assign the same number of entities to each block. Maximum of 4 reducers execute the resulting 10 reduce tasks.

Figure 14 below shows the performance when number of reducers changed for fixed window of size 10 compared against the same configuration with window of size 100. There is very slight variation in performance for JobSN and new approach when number of reducers are less but it performs very well when

number of reducers increases with window size. I observed a linear speedup for the entire range of data using the configuration of up to 2 nodes with 4 cores.

Figure 14: Comparison of existing sorted neighborhood approaches with the new suggested approach

There is a very slight difference in runtime of the different implementations. I observed differences for a very small level of parallelism. In the sequential case, RepSN was 3 minutes slower for w = 100. For the configuration (map = 2 and reduce = 2) RepSN completed way faster than new approach and JobSN. For the bigger level of parallelism, window size has no significant effect on all the 3 approaches in terms of comparative execution time, but the overall time for execution has increased with window size.

Degree of Data skew:

As explained earlier, data skew is one of the factors to be considered while evaluating the performance. In the case of higher data skew, execution time goes higher irrespective of the number of reducers assigned. The selection-skew can make an impact on execution time as it depends on the attribute value selection

for the blocking key which can also be called as intrinsic skew. In this algorithm, it impacts on selectivity skew as partitioning determines the reduce operation.

1. Support for load balancing strategies against data skew is very weak in the case of all three approaches and dependent on partitioning of data.
    a. I generated different sizes of blocks by modifying the key and Partitioner [2] function.
    b. The combination of blocking key together with key-based partitioning may create partitions of largely varying size so that a single or few reduce tasks dominates the total execution time.

    For example , if imagine in two blocks with 25 entities and a Partitioner [2] function with key values even or odd there can be various combinations based on selection and assignment of key values.(partition 1 can have 40 entities and partition 2 can have 10 entities or it can be evenly distributed among 2 partitions).

2. I calculated the average execution time for different data skews.

    After calculating the average execution time for Sorted Neighborhood with MapReduce, JobSN and RepSN approaches, I came to a conclusion that all these runs very well for evenly distributed partitions. JobSN and RepSN runs little slow in case of more number of partitions.

    In other case, where the partitions are uneven as we increased value Data-skew, performance degrades and the execution time. It was significantly higher when data partitioned with the higher value of data skew because of the effect of data skew.

The major issue identified during analysis is Data Skew, when a map operation is performed it was found that entities which are very common were grouped together and Partitioner [2] function was unable to detect the data being distributed across partitions due to lack of count. This result in

1. Increase in large variation in Size of partitions
2. Increase in number of longer reduce operations.
3. This will increase for larger window sizes since only one reducer compares more entities within a partition.
4. This can also cause memory bottlenecks if the data skew is too high

Below is the execution time when I ran the experiment on different data-skews, and below Figure 15 explains the execution time for JobSN and RepSN. It shows clearly that there is an increase in execution time when data skew is high. Here, window size kept as 10 and blocking keys were different in each execution to attain the data skew of desired value.

Figure 45: JobSN Vs RepSN execution time against Data-Skew

In Figure 16, we can see comparison of performance between all three approaches including JobSN and RepSN. After assuring that all the setup configuration which is the base setup as explained in the experimental setup section, I executed all these algorithms against different data skew values which is same for experiments carried out for the performance testing of JobSN and RepSN. Now, in Figure 16, if we compare these two approaches with the new approach then, for normal data skew, it performs equivalent to JobSN and for higher data skew it performs better than both the approaches though there is short overhead of repartitioning.

Figure 16: Performance of all approaches against data-skew

In this approach, data repartitioned only when the size of partition is more than 3 times of average partition size and hence, it does not take any step until the time it finds anything above the threshold. This helps more in case of higher data skew where execution for reduce function can take more time than the repartitioning. This technique can also help to resolve the memory bottleneck issue where reducer may need to take all the data in a partition to perform reduce operation. Due to partitioning, it does not need to take all the records. This approach is again takes advantage of the second MapReduce job functionality to complete the sorted neighborhood implementation and not skipping boundary entities.

CHAPTER 7

Conclusion and the Future Work

In this project, I attempted to improve and test the sorted neighborhood implementation approaches mentioned by Dr. Kolb and his colleagues in his paper [1]. This new improved approach is a suggested solution to a problem addressed by Dr. Kolb in his paper [1], which is about the load balancing approach. New suggested solution works better as compared to suggested approaches when data skew is very high and proves to be robust against memory bottleneck problems as well. The algorithm still has a scope of improvement as needed to handle different types of issues. One of those issues to list out would be removal of duplicate entities. In addition, this algorithm has some advantages and disadvantages. This adds extra overhead of partition in case of smaller datasets, which is very rare in case of the big data world. In the future, we hope to combine the implementation of this project with another project directed by Dr. Tran that uses the approach of entity matching as its input to return highly relevant entities.

LIST OF REFERENCES

[1] Kolb, L., Thor, A., Rahm, E.: Parallel sorted neighborhood blocking with MapReduce. In: BTW, pp. 45–64 (2011).

[2] Apache Software Foundation. Hadoop. http://hadoop.apache.org/mapreduce/, 2006.

[3] Apache Software Foundation. Spark. http://spark.apache.org/, 2014.

[4] Entity Matching For Big Data, Dedoop, http://dbs.uni-leipzig.de/dedoop

[4] Wikipedia, 'Java (programming language)', 2015. [Online]. Available: https://en.wikipedia.org/wiki/Java_(programming_language). [Accessed: 27- Nov- 2015].

[5] Wikipedia, 'IntelliJ IDEA', 2015. [Online]. Available: https://en.wikipedia.org/wiki/ IntelliJ_IDEA. [Accessed: 23- Nov- 2015].

[6] Dhruba Borthakur. The Hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.

[7] Carlo Batini and Monica Scannapieco. Data Quality: Concepts, Methodologies and Techniques. Data-Centric Systems and Applications. Springer, 2006.

[8] MapReduce: Simplied Data Processing On Large Clusters Jeffrey Dean and Sanjay Ghemawat Google,Inc

[9] Slideshare: http://www.slideshare.net/savioursofpop/entity-matching-on-social-networks-for-unilever. May 2014